SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(6), 677–692 (JUNE 1993)

Emulating an Operating System on Top of Another

MICHAEL FRANZ
Institut für Computersysteme, ETH Zürich, CH-8092 Zürich, Switzerland

SUMMARY

In this paper, we present the design of an operating-system emulator. This software interface provides the services of one operating system (Oberon) on a machine running a different operating system (Macintosh), by mapping the functions of the first onto equivalent calls to the second. The construction of this emulator proceeded in four distinct phases, documented here through examples from each of these phases. We believe that our four-phase approach can be beneficial whenever a larger software system needs to be adapted from one architecture onto another. In conclusion, we relate some of the lessons learned and propose guidelines for similar engineering projects.

KEY WORDS Software engineering Operating systems Software portability Oberon Macintosh

INTRODUCTION

Oberon[1,2] refers simultaneously to a modular, extensible operating system and an object-oriented programming language developed for implementing it. Both were conceived originally for the *Ceres*[3] personal workstation, but have since been implemented on a number of other machine architectures using a portable compiler front-end[4] as a common starting point.

This paper is based on the experiences we gained while porting the *Oberon Operating System* (subsequently called *Oberon*) onto the *Apple Macintosh*.[5] All of our examples come from this project. However, rather than giving a detailed description of our implementation, we present typical problems materializing in such a design and a methodology for solving them. We therefore do not assume familiarity with either operating system, but point out the two systems' particular characteristics wherever necessary.

In many aspects, the Macintosh operating system is similar to Oberon. Both offer relatively few but powerful operations to the programmer, as opposed to other operating systems that offer a multitude of simple operations. From another angle, however, the two operating systems are very different. Oberon is much smaller than the Macintosh operating system and attempts to present an orthogonal programming interface, in which there is usually only one way of achieving a certain task. The Macintosh operating system, on the other hand, apparently tries to be all things to all people, allowing different ways of reaching the same goal and offering heavily

0038–0644/93/060677–16$13.00

Received 22 November 1991
Revised 5 December 1992

parametrized procedures. This parametrization introduces overheads and impedes performance.

One might expect that implementing a small operating system on top of a more extensive one should be straightforward. In our experience, this is not the case. Some of the simplest functions of the emulated operating system turned out to be surprisingly cumbersome to replicate in the target system. For example, a fundamental display operation in Oberon is the procedure Dot(x, y, mode, colour), which changes the appearance of an individual pixel on the screen. Several higher-level functions are based on this procedure. But, curiously, the Macintosh operating system has no elementary operation for drawing single pixels; its manufacturer recommends use of the built-in line-drawing routine with a line-length of 0 and a line-thickness of 1 instead. This routine is needlessly complex for our purpose and requires additional parameters (such as a clipping region and a fill-in pattern), which make little sense when operating on single pixels.

We have implemented an emulator for the Apple Macintosh that presents the interface of Oberon to client programs, enabling users to bring existing Oberon software to the Macintosh by mere recompilation. This emulator, named *MacOberon*, runs as one of several processes on top of the existing Macintosh operating system and is integrated into its architecture, allowing Oberon to be used concurrently with other applications and to exchange data with them. In the following sections, we shall present our approach to this design, illuminate specific solutions, and attempt to draw some guidelines. Our experiences may be helpful to other engineers engaged in similar projects.

BARRIERS TO PORTABILITY

The major hurdles that have to be overcome in order to emulate one system on top of another can be classified under the following headings:

1. *Incompatible paradigms.* This is the most severe type of obstacle to be overcome in porting any software system, as it forces the designer to violate rules of the target system to accommodate the new paradigm. As an illustration, Oberon is based on the paradigms of decentralized control and unlimited extensibility through the addition of modules that can be loaded dynamically. The Macintosh operating system, on the other hand, is not freely extensible and is based on the concept of an 'application program', which distributes control to sub-parts that are statically linked to it. Extensibility was not anticipated by the designers of the Macintosh operating system and cannot be built into it without breaking some of its rules.
2. *Contrasting abstractions.* Less troublesome than a clash of fundamental paradigms, but difficult to bridge nevertheless, is a divergence in the central abstractions upon which systems are founded. When a certain concept in one system has no exact counterpart in the other, the designer is forced to redistribute functionality. In most cases, this involves the reconstruction of higher-level abstractions from lower-level ones. For example, Oberon differentiates between the abstractions of a data file and a mechanism for access to it, while the Macintosh operating system mixes the two concepts, associating a position with every file. Basing Oberon's abstractions on those of the Macintosh is not possible without programming some of their functions anew.

3. *Implementation restrictions.* Unpleasant for an implementer are restrictions that are present in the target system but have no equivalent in the system being simulated. They can generally not be circumvented, but moderated at best. For instance, the Macintosh operating system limits the number of files that may be open simultaneously, whereas Oberon recognizes no such restriction. The restriction can be lessened so that it will not much disturb users of the Oberon emulator, but never removed completely.
4. *Performance bottlenecks.* Least important, but not negligible, are impediments to adequate performance of the emulated system that are inherent in the design of the target system. These bottlenecks are often not at all perceived as such on the target machine, because they apply to services that are seldom used. For example, the routine that displays single characters on the screen is not efficient on the Macintosh, but there is another routine that can display whole strings. Unfortunately however, Oberon does not feature the complex operation but only the simple one, relying on its efficient implementation. Circumventing this bottleneck can increase the speed of text-display operations significantly.

Our project of implementing the Oberon system interface on top of the Macintosh operating system followed the order of these four phases, solving the fundamental problems first and settling minor inconveniences last.

We started by *identifying the major paradigms* that needed to be *adjusted* in the transition from the Macintosh operating system to Oberon, and implemented this adaptation. We then *introduced the major abstractions* of Oberon into the Macintosh framework. From this point onward, we were able to use Oberon on our target machine, although our implementation was far from being complete. As a next step, we *eliminated or weakened restrictions* of the target operating system that 'shone through' into the emulated Oberon world too prominently. At the last step, we *fine-tuned performance* by singling out performance bottlenecks and trying to work around them.

The following sections examine in turn the different phases of our project by giving representative examples of the problems encountered. We believe that the same four-phase approach can be applied to many other projects that involve adapting software from one base to another.

PHASE ONE: ADJUSTING PARADIGMS

As we have stated above, changing a paradigm in an existing system is a grave assault on the integrity of that system. This step must be executed with the greatest care and is not possible without violating existing rules, making the modification 'unsafe' from the viewpoint of the unmodified system. Equally critical in practice is the fact that there is usually no documentation available that explains how to carry out these 'illegal' alterations, or how to minimize their potential danger.

In order to reproduce the structure of Oberon on the Macintosh, we had to make two such fundamental adjustments to the run-time architecture of the Macintosh. The first one was necessary for the support of dynamic module-loading, and the second for duplicating Oberon's control flow, specifically the facility to abort running commands.

Adding unlimited extensibility

Application programs on the Macintosh are event-driven. Their behaviour from moment to moment is determined by input from the user in the form of mouse and keyboard actions. Whenever such a user action occurs, the Macintosh operating system makes a record of it in a so-called 'event queue'. There are several such event queues, one for each running application, and the operating system uses a heuristic method for assigning events to specific event queues. For example, keyboard events are forwarded to the application that owns the front-most window on the screen. In the Macintosh co-operative multitasking environment, different application programs gain control every so often and then process the event records in their respective event queues asynchronously.

The overall architecture of Oberon bears some similarities to that of the Macintosh operating system. Unfortunately, however, our Oberon emulator needs to act as an *application program* on the Macintosh, which requires a markedly different internal organization (Figure 1). Each Macintosh application is structured around a central

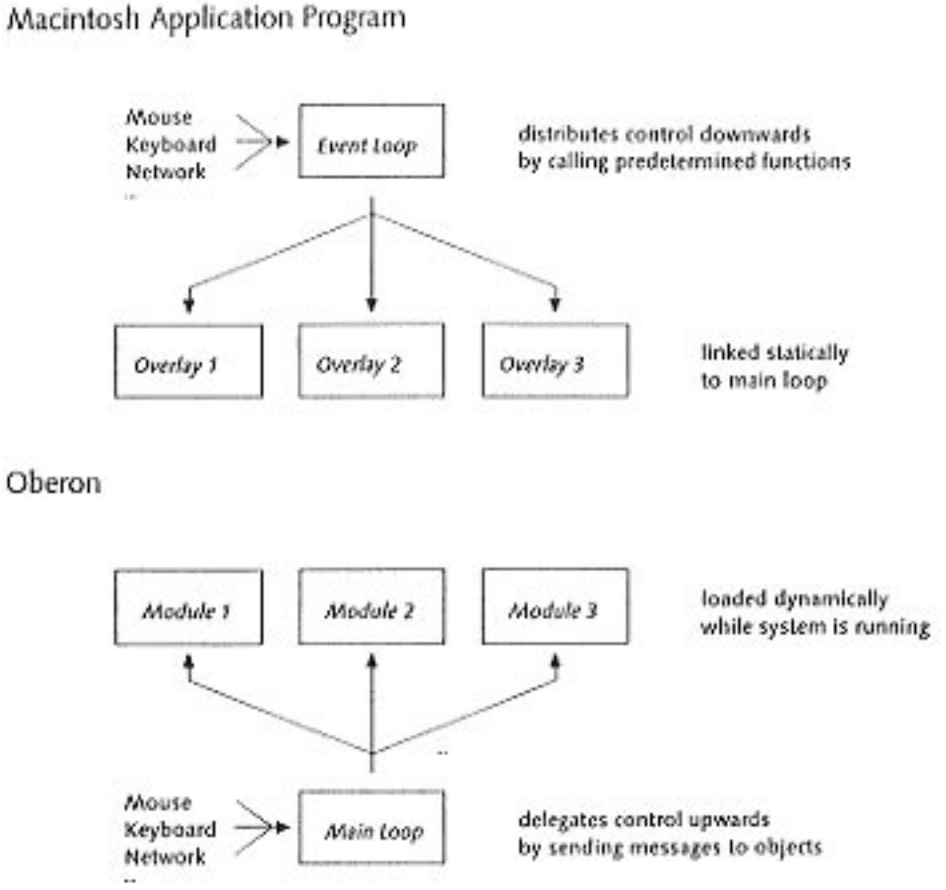

Figure 1. System architecture

'event loop', which extracts events repeatedly from its private event queue, analyses them, and executes specific actions in response. Since the correspondence between particular events and resulting actions is hard-coded into it, such an application cannot be extended to include new functionality without changing its program source.

Oberon, on the other hand, is extensible. Its 'main loop' does not associate any specific actions with certain classes of events but instead responds to them by sending 'input messages' to objects representing viewers (windows) on the screen. The action taken in response to a user input depends on the particular viewer that receives the corresponding input message. Viewers are implemented in program modules that are positioned hierarchically above the main loop. These modules are loaded dynamically while Oberon is running. Extending the system is easy and done by creating new classes of viewers that implement new functions. Adding new classes of viewers to Oberon is more or less analogous to writing new application programs in other operating systems.

Incorporating the extensibility of Oberon into an application program cannot be accomplished within the rules of the Macintosh operating system because it requires a facility for the dynamic loading of additional program modules while the application is already running. Although the Macintosh operating system offers a mechanism for dynamically loading parts of a program as overlay segments, this mechanism cannot be employed for our purpose since it uses a global, fixed-size link table for redirecting calls between different segments. Using overlay segments for the simulation of dynamically-loadable modules would require a unique numbering of modules and would preclude the addition of further modules after all link-table slots were filled.

Consequently, we had to construct a new module loader for our Oberon emulator on the Macintosh. This module loader itself is written in Oberon and installed by a so-called 'boot-loader'. Only the boot loader is a 'proper' Macintosh program recognized as such by the Macintosh operating system. All other modules subsequently loaded are considered to be 'data'. Nevertheless, the instructions contained within these modules are executed after loading, violating a convention that strictly separates data from executable code. Luckily, this convention cannot be enforced by the Macintosh operating system, owing to the fact that it does not support memory protection.

Changing the flow of control

The Macintosh event queue contains more than just simple input events that can be processed directly by Oberon's main loop. There are also meta-events that concern the emulator application as a whole, e.g. the 'suspend' and 'resume' events of the Macintosh co-operative multi-tasking environment, administrative events instructing an application to redraw parts of the screen, and events relating to the standard Macintosh user interface. All of these need to be processed on the 'Macintosh side' of the Oberon emulator, as the emulated Oberon system has no equivalent concepts.

Emulating Oberon within a Macintosh application program requires moving the event loop from the top of the module hierarchy almost to its bottom. In our Oberon emulator, the Macintosh event queue is processed as a side-effect of Oberon's keyboard polling. Each time that MacOberon inspects its keyboard buffer,

which does not exist in the Macintosh operating system and is therefore simulated, the Macintosh event queue is scanned and all pending keyboard events are accumulated into this buffer. Meanwhile, all event records that have no match in Oberon are extracted from the event queue also and processed directly (Figure 2). Control is returned to the emulated Oberon system after all events in the queue have been processed.

Regrettably, however, Oberon's main loop not only delegates control upwards, but can also take it back at any time, in response to a special user-break command from the keyboard. This pre-emptive break mechanism cannot be integrated into the Macintosh environment so easily. The usual way of sending a signal to a running application on the Macintosh is via the event queue. However, we require pre-emption at arbitrary times, not only at the points at which they access this queue. MacOberon therefore instals an interrupt routine that is executed several times per second and monitors the keyboard for the simultaneous depression of a certain key combination. When this condition is detected, steps are taken that will return control to the Main Loop as soon as this can be done safely.

A running command is aborted by temporarily substituting a supervisor call in

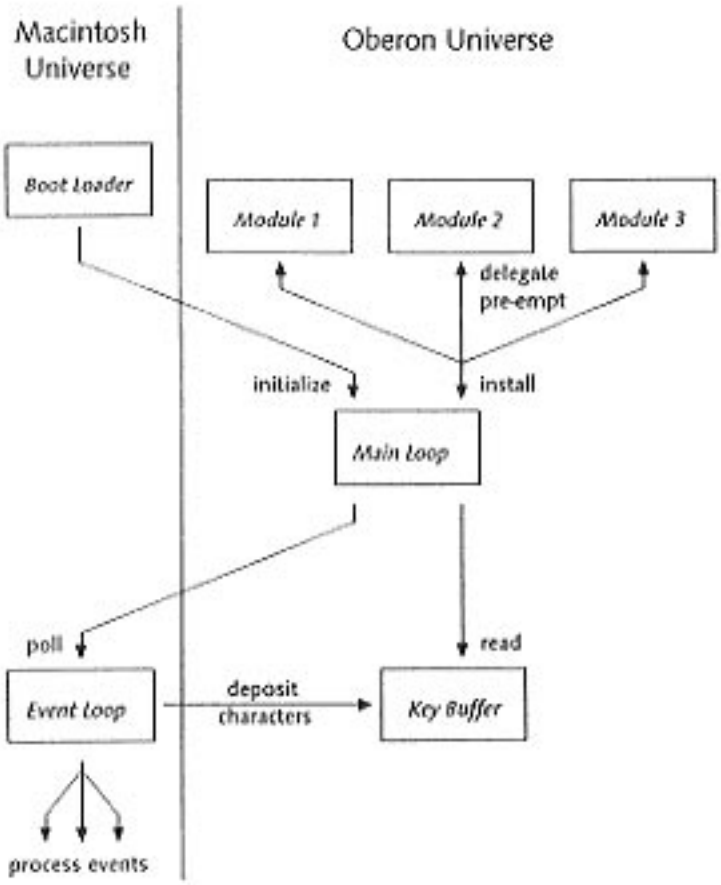

Figure 2. Flow of control in MacOberon

place of an instruction that is just about to be executed. However, this is not as straightforward as it may seem because user breaks are detected inside an interrupt service routine, at which point the calling chain may contain routines of the Macintosh operating system that must always return normally. Moreover, there are certain low-level routines of the Oberon emulator that access global resources and that should therefore not be pre-empted either. Consequently, our break mechanism needs to examine activation records on the stack, locating the topmost procedure in the calling chain that can safely be aborted and inserting a break instruction only there.

Pre-emption conflicts strongly with the usual programming model on the Macintosh, in which every subroutine always returns to its caller. From the viewpoint of the Macintosh operating system, the control flow in the Oberon emulator is 'unsafe'. However, MacOberon can guarantee the safety of this mechanism because it keeps track of the entry and exit points of every procedure. It can, therefore, identify the owner of every activation record on the stack uniquely and ensure that critical operations are completed before the flow of control is redirected.

PHASE TWO: MAPPING ABSTRACTIONS

Systems are built on a small set of abstractions, which are usually accessible to programmers through an interface consisting of a few abstract data types and operations on these types. Constructing such a high-level interface from another high-level interface can be a difficult task, more difficult than it would be to build upon a low-level interface.

A one-to-one correspondence between abstractions that can be mapped directly onto each other is the exception rather than the rule. In typical scenarios, m abstractions on one side need to be expressed by n abstractions on the other. In the following, we shall study two examples from our implementation that represent the two extremes of many-to-one and one-to-many mapping (Figure 3). The treatment of other configurations is analogous and involves a combination of the strategies discussed here.

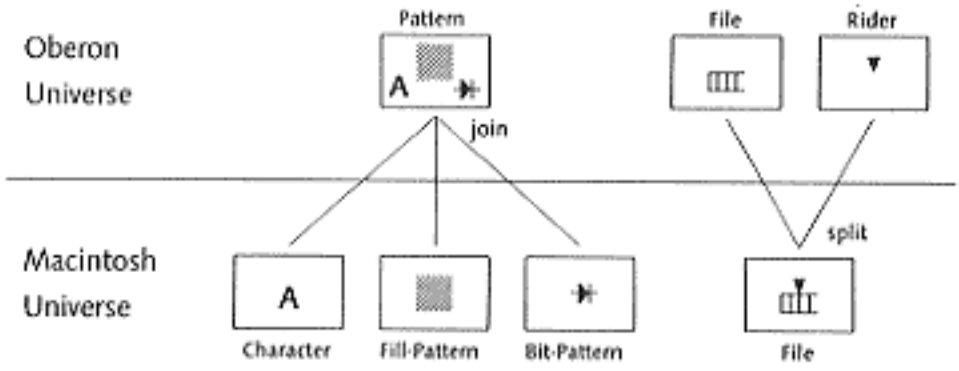

Figure 3. Abstraction mapping

Merging abstractions: patterns

Oberon bases its imaging model on a single abstraction, called a 'pattern', which is a bit-arrangement that may be painted or replicated on the screen in one of several colours and transfer modes. All graphic elements that can be displayed in Oberon are based on patterns, e.g., characters, cursors and grey-scales.

The Macintosh graphic system, on the other hand, uses several different abstractions side by side. There is a data structure describing images that are used for filling in areas in a regular fashion, and another one describing less regular bit-patterns that can be painted only, but not replicated. The raster data of characters are not accessible at all on the Macintosh. Instead, there are dedicated routines for drawing characters and character strings. For the purposes of emulating Oberon, we had to hide these three different mechanisms behind the single pattern abstraction.

The unpleasant distinction between replicatable and non-replicatable images on the Macintosh side was overcome by *translating* each Oberon pattern into both Macintosh representations, which could then be kept simultaneously in a single data structure. The abstract data item that represents the pattern within the emulated Oberon system is a pointer to this structure. Depending on the display operation selected, one or the other variant is used when the call is forwarded to a routine of the Macintosh operating system.

Characters are treated differently yet again. The Macintosh displays characters on the screen by using dedicated text-drawing routines, which require the specification of parameters such as the character's font, size and style attributes besides its ordinal value. In analogy to the two other variants of patterns, we could have represented the members of the third pattern species also by (pointers to) descriptors in which their characteristic attributes were stored. However, this would require a separate descriptor for each character in every font being used. Character patterns can be represented much more efficiently by encoding the character-code information directly in the pattern (Figure 4). A distinction between directly-encoded patterns (describing characters) and storage-allocated patterns (describing other kinds of

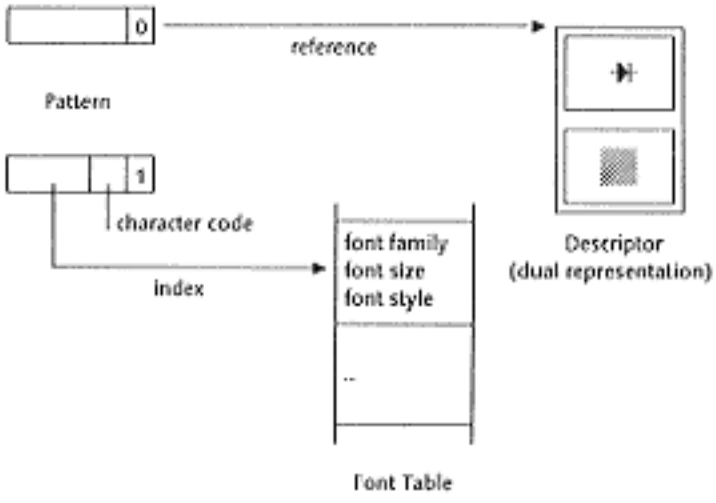

Figure 4. Representations hidden behind Oberon's pattern abstraction

images) is easily found by using odd values for the former; the latter are pointers, which are always even on the Macintosh.

In this fashion, three different concepts on the Macintosh could be unified in the single Oberon abstraction of the 'pattern'. Only the emulator's display routines make explicit use of the fact that there are three different varieties of patterns, among which they distinguish automatically, and quickly. As far as any other Oberon routine is concerned, a 'pattern' always describes an image, be it a character bit-map, a more general picture, or a replicatable shade of grey.

Splitting an abstraction: files and riders

The Oberon system differentiates between the abstractions of a *data file* and an *access mechanism* to it, which is called a 'rider'. An arbitrary number of these riders may operate concurrently on the same file, sharing buffers if their ranges overlap. In contrast, the Macintosh operating system mixes the two abstractions, making the 'current position' a property of the file. Several access paths to the same file may be created by 'opening the file several times'. However, each of these paths uses its own private buffers and the integrity of the associated data file may be destroyed by overlapping buffers.

In order to map one interface onto the other, one has to *split* the 'file' abstraction of the Macintosh operating system. Splitting abstractions is far more troublesome than merging them. It usually involves a substantial amount of programming when the resulting abstractions cannot be constructed directly from the ones to be split, but have to be reassembled from lower-level ones.

Oberon's concept of a 'file' is almost fully contained in the 'file' abstraction of the Macintosh operating system, whereas its notion of a 'rider' is very different from the 'multiply open file' mechanism. Not surprisingly then, implementing Oberon's 'files' on the Macintosh was relatively easy, whereas implementing 'riders' was not.

In order to achieve the former, all we had to do was to separate the directory operations from the file operations. Oberon distinguishes between the two, allowing several files of the same name to exist concurrently, only one of which is registered in the directory. At any given moment, a directory lookup will return the instance of the file that was last registered. This behaviour can be simulated by a naming scheme that assigns temporary names to files that are not registered in the directory and by renaming files when necessary. All of these non-registered temporary files are deleted at the end of a computing session.

The 'rider' mechanism, on the other hand, had to be reimplemented from the buffer level upwards, employing the services of the underlying operating system for the transfer of integral buffers between memory and secondary storage only. As a beneficial side-effect of this reimplementation, the routines offered by our emulator are faster than their counterparts of the Macintosh operating system, because they are less general, because the overhead for calling them is smaller, and because they employ a higher degree of buffering, which is sensible when memory abounds as it does on today's computers.

PHASE THREE: LIFTING RESTRICTIONS

Some restrictions are engineered so deeply into an operating system that it is not possible to eliminate them at reasonable expense. However, it may still be worthwhile to try to mitigate their effects if a balance between cost and result can be found. Our example will show how we were able to diminish the effects of an important limitation considerably, without having to invest much effort, because we could build upon features that were present in our implementation already.

Increasing the number of files in use

The Macintosh file system restricts the number of files that can be open simultaneously on one machine, i.e. by all active applications combined. This number is unfortunately quite small, of the order of tens rather than hundreds in typical system configurations. Oberon, on the other hand, makes heavy use of files and allows any number of them to be active at the same time. In the course of typical Oberon sessions, large numbers of files are open concurrently, many of them anonymous temporary files that are needed for a short time only, but which are neither explicitly closed, nor deleted. On native Oberon machines, this is natural because disk space is reclaimed only when the system is started, by a garbage-collection process.

We were able to diminish the restrictions that the Macintosh operating system imposed on our emulator by applying two countermeasures. First, we recognized that the majority of files in Oberon were newly-created ones, which were being used as all sorts of temporary buffers. These buffers were typically quite small, of the order of several hundred bytes. It was therefore logical to defer the physical creation of new files until their length had exceeded a certain limit or until the user registered them in the directory in order to make them permanent. By keeping small new files in memory completely, we were able to reduce the use of disk-based files to a minimum, so that the limitations of the Macintosh operating system would scarcely ever be reached. This was not difficult to implement, as we had already programmed our own buffering in order to support multiple riders.

As a second action, we optimize the use of available file-access paths by keeping files open only as long as absolutely necessary. A file can be flushed to disk and closed as soon as it can be guaranteed that it will no longer be used. The latter is the case when all references to the file become unreachable, which is information that can be obtained as a by-product of garbage collection. Since Oberon always includes a garbage collector, we were able to incorporate automatic file-closing at very little additional expense.

PHASE FOUR: REMOVING BOTTLENECKS

Any piece of software can be optimized to run as efficiently as possible under one specific implementation of an operating system. This is done by making intelligent use of the resources offered, giving thought to the price of each operation and minimizing total cost. However, when the same environment is simulated on another architecture, the relative cost of the individual services may shift dramatically, and optimized programs may actually run slower than non-optimized ones.

In our case, we were dealing with software that had been optimized to run under Oberon. In fact, since Oberon has a small and orthogonal interface, the

straightforward solution is usually already the optimal one for Oberon. But now this software was required to run on the Macintosh, indirectly using the services of a less regular operating system, in which sophisticated functions often yield performance far superior to their simple counterparts.

The following example will show how we were able to make use of the more complex functions of the Macintosh operating system transparently, increasing the performance of all Oberon programs simultaneously without requiring a change in each for accommodating the more complex interface of the Macintosh.

Accelerating text-display operations

The Macintosh display system is far more hardware-independent than that of Oberon. It supports a wide range of display devices, differing in size, resolution, depth and even pixel arrangement. Several different pieces of display hardware can be used concurrently and display operations may involve more than one device. Consequently, the operating system has to break down every activation of a display routine into separate calls to device-specific operations for each of the devices involved.

Naturally, this generality is very costly. When drawing individual characters onto a single display device, the overhead of each call is close to the actual effort required to copy the character's raster data into the frame buffer. Typical Macintosh applications use string-drawing routines instead of character-drawing routines, which require an overhead only once for a whole sequence of characters. However, Oberon's orthogonal interface features a character-drawing routine only, which has to be fast, as it is the one factor that influences responsiveness the most, and is therefore critical to user satisfaction.

Adequate performance of Oberon's text routines on the Macintosh was achieved by way of a string buffer. As we have explained above, the routines that draw Oberon's patterns onto the screen can distinguish whether these describe bit-images, grey-scales, or character glyphs. Furthermore, from patterns describing characters they are able to reconstruct the ordinal value of the original character, as well as its font, size and style attributes. This information is used to assemble sequences of characters that share the same attributes, which are then drawn as a string in a single operation. Two characters with the same attributes can be considered part of a string, if the second is positioned at the same vertical position as the first, and at a a horizontal position offset from the first character's horizontal position by that character's width.

In our Oberon emulator, drawing operations involving characters are delayed while the display subsystem attempts to build strings of characters. The string cache is flushed periodically, and is guaranteed to be empty before any other drawing operation is executed that could be in conflict with caching (part of a string whose drawing is delayed might, for example, lie within the source range of a block move operation). By the use of string caching, we were able to boost performance by a factor of more than 10 in typical situations, making it comparable to that of programs using the string routines of the Macintosh directly.

LESSONS LEARNED

It is impossible to pass on completely the wealth of experience that we have gained during our implementation. While our overall design proceeded in four phases as we have outlined above, individual design decisions in each of these phases were governed by a yet-unwritten set of rules, which come under the general heading of 'engineering common sense'.

Nevertheless, in the course of our design we ran into dead-ends repeatedly, and had to undo and rebuild. In hindsight, we would like to formulate the lessons that we have learned, the essence of 'engineering common sense', so to speak, into seven general guidelines. As we have done above, we shall present illustrations from our project for most of these rules. The reader should keep in mind that in reality there are many complex interrelationships, so that more than one rule may be applicable for each of the examples given.

1. Develop on the target machine

In our experience, development on the target machine is more productive by at least a factor of two, because turnaround times are much shorter. Additionally, having to depend on the correctness of one's own programs is a healthy lesson indeed. Whenever our Oberon editing environment crashed during development on the target machine, invalidating hours of work, there was nobody but ourselves to blame. The program editor that we used in our development, the Oberon routines called by it, and the compiler that was used for creating it, are now among the most stable components of Oberon on the Macintosh, because serious errors were discovered during the early stages of development.

2. Optimize later, if you have to

If you have to 'improve' on established practices, save this for a later phase of your project, unless it is absolutely necessary for accomplishing your goal. Many potential optimizations will turn out to be redundant later, but drown you in their complexity when you attempt to do all things at once.

For example, the compiler we developed initially for our Oberon project optimized data alignment. Incorporating this feature cost us a lot of time and effort, at a time when we were still cross-developing. By supporting two different sets of calling conventions and data-alignment tactics, the compiler could guarantee that the stack would always lie on a long-word boundary.[6] This resulted in a performance gain of about 20 per cent on MC68020 processors, but the advantage vanished in the newer models of the processor family, owing to the effects of cache memory. Meanwhile, the optimizations added to the complexity of the compiler quite considerably.

We estimate that we would have been able to start development on the target machine one month earlier, gaining two months' worth of cross-development productivity, if we had not built optimal alignment into our compiler from the start. Ironically, the feature was dropped in a later version, reducing the code generator by about 20 per cent in size.

3. Add features transparently

In some respects, the target system may offer services that are superior to (and a superset of) the corresponding functions of the system being simulated; if these advanced features can be integrated into the emulation transparently, it is logical to incorporate them. However, transparency implies that client modules should not depend on whether they are running under an emulator or on a native system.

As an example, the Macintosh font mechanism is superior to that of Oberon, because it provides for the automatic on-the-fly generation of glyphs in any raster size from an outline description, whereas Oberon requires the explicit naming of a font file that contains the appropriate raster data.

Our implementation of Oberon on the Macintosh parses this file name to extract font name, size and style information and then requests the corresponding font from the Macintosh font manager. As long as the standard font-file-naming conventions are adhered to, this mechanism is fully transparent to Oberon applications, but gives users on the Macintosh access to all fonts available on their machine, not just those distributed with Oberon. An attempt to access one of these fonts in another non-Macintosh Oberon system will simply return the default font, which is the standard behaviour of Oberon if a certain font file cannot be located.

4. Do not prejudice

Although it is very likely that our Oberon emulator will be used like other Oberon systems on other hardware platforms, there might one day be a programmer with special applications in mind, involving, for example, the Macintosh environment around the emulator. As it is impossible to anticipate all future applications that might come along, one should always design for maximum flexibility as long as this can be done economically.

For instance, we had to implement a heap-storage allocator and a garbage collector for MacOberon. The Macintosh offers no virtual addressing but it would have been straightforward to map Oberon's heap onto a contiguous area of primary storage on the Macintosh. However, there are certain built-in routines on the Macintosh that allocate memory directly through the Macintosh operating system. Regular Oberon programs never need to access any of these routines, but what if some 'special application' running under our emulator wished to do so anyway? Support of the routines in question would require a sensible heuristic by which to leave a certain amount of memory unused for them without taking away too much from the emulator.

Luckily, there is another solution that requires no heuristic, generates almost no overhead, and gives maximum flexibility even to 'unusual' user programs. Our implementation uses a 'discontinuous heap', which consists of several medium-sized blocks that are requested from the target operating system and returned from it as needed. Within these blocks, the Oberon emulator manages a fine-grained subdivision on its own. This scheme introduces an insignificant additional effort while it allows MacOberon to occupy only as much memory as is needed, leaving the remaining storage at the disposal of the user.

5. Integrate

Although it may sound paradoxical, even when emulating a complete operating system within another, to end-users this should appear to be well-integrated into the host operating system. In our case, matters were slightly more complicated, as we wished to integrate an Oberon emulator into the operating system of the Macintosh, but then integrate this Macintosh into a local-area network of Ceres[3] computers running Oberon as the native operating system. Integration into the Macintosh operating system was achieved by sharing resources with other Macintosh applications, and by supporting certain harmonizing features of the host operating system.

For example, MacOberon does not take over the whole screen but is confined to a window coexisting with the windows of other applications. It supports the Macintosh operating system's mechanisms of co-operative multi-tasking and inter-application switching. Other aspects of integration include support of the Macintosh Clipboard, which allows text with font and style attributes to be transferred from one running application to another by 'cutting' and 'pasting', and the incorporation of the Macintosh font machinery in addition to the one usually found in Oberon.

On the other hand, integration into the Oberon network was possible by reimplementing the network protocols of the Ceres computer on the bare Macintosh hardware, bypassing the network services of the Macintosh operating system. Both machines use the same hardware interface.

6. Unify concepts

Do not implement several variants of the same mechanism, but try putting to use a single generalized version. Take the following example: like other Macintosh programs, our Oberon emulator features pull-down menus at the top of the screen, offering commands for purposes such as quitting the application and accessing the 'cut and paste' mechanism. In early versions of the emulator, the contents of these pull-down menus were fixed and the actions to be performed hard-coded into the loop that examined the event queue, just as in every other application on the Macintosh.

But then it occurred to us that the more general activation-by-name mechanism available for procedures in Oberon could be employed even for calling commands in these menus. The Oberon loader not only allows modules to be loaded dynamically at run-time, but also has a function to obtain a procedure's entry point when its name is known. Our pull-down menus contain a module name and a procedure name in a fixed syntax. When a menu item is selected, the associated string is parsed to obtain these two names. The corresponding module is then loaded dynamically and the procedure mentioned in the menu is called.

MacOberon is now the only Macintosh application known to us that allows the user to link any function that can be performed at all within the application to any item appearing anywhere in the menu bar. These functions can be added and removed without modifying the program, simply by editing a Macintosh system resource that contains the menu text. Interestingly enough, building this generality into our emulator amounted to removing lines of code, as the mechanism was present already.

7. Make it as simple as possible (but not simpler)

Einstein's rule, which has guided the Oberon Project from its very conception, fundamentally underlies all others cited here and is the most important of all. Whenever in the course of our project we had to make a critical design decision, the simplest solution turned out to be superior in the end.

The importance of simplicity and clarity of design cannot be overstressed. We were able to adapt a compiler for the Oberon language to the Macintosh in less time than it took us to implement the operating-system interface on the same machine, although a complete code-generator had to be created from scratch. Initially, we expected the construction of a compiler back-end to be far more difficult and time-consuming than that of an operating-system emulator. However, in the case of the compiler we were able to build upon the simple and concise design of the existing front-end, whereas for the emulator we had to interface to a complex, inefficient and often uselessly general behemoth of an operating system.

SUMMARY AND CONCLUSION

Emulating a complete operating system on top of another is an interesting software-design project, because it involves different classes of problems, ranging from deeply-rooted architectural issues to such mundane, albeit equally important, topics as fine-tuning of performance. By keeping the design simple, and structuring it, even projects with a wide scope such as this can remain easily manageable.

We have outlined the four-phase approach of our design. We believe that this approach can be applied in many other situations that involve adapting a significant amount of existing software to a different platform. We are further convinced that the complexity of projects such as ours can be reduced considerably by tackling individual problems in the order that we have outlined.

We have attempted to present real problems that have come up in an actual project. Probably none of our solutions to these individual problems is novel. Nevertheless, we hope that our survey of the different kinds of problems encountered has given an insight into the diversity of issues that one is confronted with when porting software systems, and wish that our experiences will benefit many readers in some way.

ACKNOWLEDGEMENTS

Régis Crelier and Josef Templ developed implementations of Oberon for other machine architectures concurrently with our project. Some parts of the design presented here were inspired by their ideas or matured during discussions with them, each having slightly different problems to tackle but all sharing the same vision. The author would like to thank Niklaus Wirth for creating Oberon and thereby giving us this vision, for initiating the project described here and guiding it through all of its stages, and for his constructive suggestions and comments regarding this paper.

REFERENCES

1. N. Wirth, 'The programming language Oberon', *Software—Practice and Experience*, **18**, 671–690 (1988).

2. N. Wirth and J. Gutknecht, 'The Oberon system', *Software—Practice and Experience*, **19**, 857–893 (1989).
3. H. Eberle, 'Development and analysis of a workstation computer', *Dissertation No. 8431*, ETH Zurich, 1987.
4. R. Crelier, 'OP2—a portable Oberon-2 compiler', *Proc. Second International Modula-2 Conf.*, 1991, pp. 58–67.
5. Apple Computer, Inc., *Inside Macintosh*, Addison-Wesley, Reading, Massachusetts, 1985.
6. M. Franz, 'The rewards of generating true 32-bit code', *SIGPLAN Notices*, **26**, (1), 121–123 (1991).